Chapter 4. RPC-Style Services

Although SOAP is not limited to a particular style of distributed computing, it lends itself to a remote procedure call (RPC) model. This is only what you would expect; according to the SOAP specification, one of SOAP's design goals is to encapsulate and exchange RPC calls. This approach maps very nicely to Java programming because method calls on Java objects can be easily translated to RPC calls. We'll start this chapter by looking at the structure of SOAP RPC request and response messages. From there we'll implement some RPC services in Java and deploy those services in both Apache SOAP and in GLUE. And, of course, we'll write some Java code to make use of those services.

4.1 SOAP RPC Elements

Creating a SOAP RPC request uses the SOAP structure and encoding described in Chapter 2 and Chapter 3. No new XML or data encoding styles are needed for RPC. Let's take a look at what's required to represent an RPC method call in SOAP:

- The target object
- Method name
- Method parameters
- SOAP header data

4.1.1 Target Object URI

The target object URI is, essentially, the resource address of the service that we want to use. You see resource addresses all the time when you browse the Web — you navigate to a web page by specifying the page's resource address. For instance, to look at the page describing the author of O'Reilly's JavaBeans book, you'd use the URL http://www.oreilly.com/catalog/javabeans/author.html. The protocol, server address, and resource are all together. Let's break it down so you can see the parts. The server address is *www.oreilly.com*, the HTML page resource is */catalog/javabeans/author.html*, and the protocol is *http*.

This is exactly what we do to specify the target object in a SOAP RPC request message, with one difference: instead of an HTML page resource, the target object name is used. The target object name is carried by the transport (for example, by an HTTP header), not the SOAP message itself. This is the only part of a SOAP RPC message that is carried outside the message except for the `SOAPAction` attribute. In an Apache SOAP server, the target object for an RPC message is usually `/soap/rpcrouter`. This URI represents the resource that routes SOAP RPC messages in Apache SOAP. For a GLUE server, the target resource is usually `/glue`, followed by the name of the service itself. We'll look at the HTTP headers for both of these servers a little later.

4.1.2 Method Name

The method name is, as you might have guessed, the name of the method to be invoked. Back in Chapter 2 we talked about a service that provided the current temperature at a weather station. That service exposed a method called `getCurrentTemperature`. If you like to think

in Java terms, you can think of the method name as a public method on an instance of a service object.

In the SOAP XML, the method being invoked is packaged as a SOAP struct[1] that contains an accessor for each parameter. The method name `struct` is an immediate child of the SOAP `Body` element. Let's look again at the `getCurrentTemperature` example:

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
       "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
         <m:GetCurrentTemperature xmlns:m="WeatherStation">
                <m:scale>Celsius</m:scale>
         </m:GetCurrentTemperature>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Notice that the name of the method being invoked is namespace-qualified using the name of the service object. The namespace is supposed to be a URI that provides the target service with a mechanism to interpret the meaning of the method. The URI would normally point to a resource that describes the method, but in this example this will work just fine. But what service object are we talking about? It's not the target object. It's another object that is known to the target object by a particular name. In this case, the service object's name is `WeatherStation`. The mechanism used to map the names of service objects to their physical implementations is not specified by the SOAP specification. Each SOAP implementation has its own way of doing that, which we'll see in Section 4.3.

The method signature may also be included as part of the method struct, but it isn't required to be there. In fact, the SOAP specification doesn't describe the format of the method signature, and signatures won't be used in this book (and aren't used by the servers I discuss). I mention method signatures only so you won't be confused if you run into an implementation that uses them.[2]

4.1.3 Method Parameters

The parameters to the method are modeled as accessors and are immediate child elements of the method struct. This is true for all `in` and `in/out` parameters; any `out` parameters will not appear in the method invocation struct. SOAP allows for the processing of methods with missing parameters, as we saw in Chapter 3. It's up to the application to decide whether or not to process a call with missing parameters. If leaving out parameters is not acceptable, the application should respond with a SOAP fault. Each accessor is named after the parameter name for the enclosing method, and its type is the same as the corresponding service method type.

The previous SOAP envelope has one parameter to the `getCurrentTemperature` method. The parameter is called `scale`, and is used to specify the temperature scale that should be

[1] The term `struct` is used here to mean the XML element hierarchy used to represent a service method and its parameters. For all you C programmers out there, don't confuse this with a C struct, which obviously has a different meaning.

[2] I doubt that you'll ever encounter a method signature as part of the SOAP message. There doesn't seem to be any industry acceptance of the concept, and it isn't implemented by any SOAP engines I've encountered.

used to return the result. This parameter is namespace-qualified using the service name, which is not unlike using a schema name to qualify the accessor. This means that the `scale` element is presented according to its definition in the `WeatherStation` namespace. We could have chosen to include an `xsi:type` attribute on `scale`, but omitting the attribute is valid as well. As long as the type matches the type that's specified in the schema referred to by the namespace, you don't have to specify the type in the SOAP message. You can even send a different type, as long as you specify what the type is.

Let's look at some Java code and see how it maps to a SOAP RPC method call. The `DowJones30` class represents the current levels of the Dow Jones industrial average. `DowJones30` contains a method called `getCurrentLevel( )` that returns the Dow Jones industrial average for the current trading session. It also has a method called `getPrice( )` that returns the current trading price for a member stock of the Dow Jones Industrial Average. `getPrice( )` takes a string parameter called `stock` that represents the stock symbol, and a `currency` string parameter that specifies the currency that should be used when returning the price. The price is returned as a `float`.

```
public class DowJones30 {
   private float _level;
   public float getCurrentLevel(  ) {
      return _level;
   }
   public float getPrice(String stock, String currency) {
      float result;
      // determine the price for stock and return it
      // in the specified currency
      return result;
   }
}
```

Now let's say we've already deployed a service named `DowJones` that is implemented by the `DowJones30` class. To invoke the `getCurrentLevel( )` method of the `DowJones` service, we use a SOAP envelope that looks like this:

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
       "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
         <m:getCurrentLevel xmlns:m="DowJones">
         </m:getCurrentLevel>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

There are no parameters, since the `getCurrentLevel( )` method does not take any parameters. Now let's call the `getPrice( )` method, using the stock symbol `IBM` and a currency of `USD` (U.S. dollars) as parameters. Here's what the envelope would look like:

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
       "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
         <m:getPrice xmlns:m="DowJones">
            <m:stock>IBM</m:stock>
            <m:currency>USD</m:currency>
         </m:getPrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

4.1.4 Method Response

A method response also takes the form of a SOAP struct. The method struct contains an accessor for the return value, followed by accessors for each `out` and `in/out` parameter value in the same order they appear in the method signature,[3] if one exists. The names of the accessors for the `out` and `in/out` values correspond to their names in the method signature, but the names of the return values don't matter. My preference is to give the return value a name that reflects the method name itself. I tend to name service methods using a verb-noun style, which then allows me to name the return value accessor with the noun. For instance, if my service name is `computeSize`, then I name the return value accessor `size`.

It's become common practice to name the return value struct by appending the word `Response` to the name of the method that was invoked. So for a method called `computeSize`, the struct containing the result would be named `computeSizeResponse`. Although this is a common convention, it is not required, and you'll find that some SOAP implementations don't follow it.

Here's the SOAP envelope that might be returned by invoking the `getCurrentLevel` method:

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
       "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
         <m:getCurrentLevelResponse xmlns:m="DowJones">
                <m:currentLevel>10513.15</m:currentLevel>
         </m:getCurrentLevelResponse>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

4.1.5 Fault Response

If a SOAP RPC method call fails, a SOAP fault should be returned. Let's take a look at an example of a SOAP fault. Say we call the `getPrice` method of the `DowJones` service but specify an invalid currency parameter. The service might return a fault that looks like this:

[3] As mentioned earlier, the method signature referenced in the SOAP specification does not appear to have been implemented. It is mentioned here for your information only.

```
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV=
      "http://schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle=
      "http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Body>
     <SOAP-ENV:Fault>
        <faultcode>SOAP-ENV:Client</faultcode>
        <faultstring>Client Error</faultstring>
        <detail>
           <m:dowJonesfaultdetails xmlns:m="DowJones">
              <message>Invalid Currency</message>
              <errorcode>1234</errorcode>
           </m:dowJonesfaultdetails>
        </detail>
     </SOAP-ENV:Fault>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

A SOAP RPC method call either fails or succeeds, but never both. So you'll never find a method response and a fault response together.

4.1.6 Optional Header Data

Processing an RPC sometimes requires information that is not part of the method signature. For instance, it may be necessary to pass a username to a service. A username is usually not appropriate as a parameter to the method call, but nonetheless is needed by the receiving application. Information like this can be encoded in the SOAP header, as described in Chapter 2. There can also be transport-specific data; for example, you may want to pass cookie information as part of your HTTP header in order to specify the HTTP session to use for this method invocation. In either case, this additional data is encoded outside the bounds of the SOAP body.[4]

4.1.7 Service Activation

The lifetime of an object is an important part of any distributed system, and is sometimes referred to as *service scope* or *service activation*. Whatever terminology is used, it encompasses the various possible object lifetime models available in the system. In SOAP systems based on HTTP, there are generally three service activation models:

Request-level service activation

> For each request made on a given service object, a new instance of the object is created that persists only until the request is complete. This kind of service activation is useful for service objects that don't maintain state.

Application-level service activation

> A single instance of the service object handles every method invocation for the entire lifetime of the service application. This kind of activation can be used for services that maintain state or that place a large burden on resources during object creation.

[4] Actually, sending transport-specific data such as a cookie is probably not such a good idea, because it binds you to a particular transport. Some other transport-independent session data would be more appropriate.

Session-level service activation

A single instance of the service object is used for all method invocations for the lifetime of the session. Sessions are not necessarily a standard, as you may know. It's possible to implement sessions specific to your own applications by doing things like generating and passing tokens, or you can take advantage of existing approaches like the use of cookies. This is certainly one of those cases where you need to pass additional header information.

The SOAP specification doesn't define any requirements or behaviors for service object lifetime; there are no real requirements for SOAP implementations to support the three models described here. However, these activation models are found outside of SOAP as well, and you can expect to find them in many if not all of the SOAP implementations available. Each implementation may have its own unique way of setting the activation model for a service, but the behavior should be the same. You'll probably find that the service activation model for a service is specified as part of the service deployment. We'll cover this topic in practice when we talk about service deployment later in this chapter.

4.2 A Simple Service

Let's design a simple service called `CallCounterService`. This service keeps track of the number of method calls it receives. It's not exactly useful by itself, but it gives us a chance to get a service up and running to see how services are deployed and how the service activation models work. We'll place the classes for this example in the package `javasoap.book.ch4`. Make sure to create the appropriate directory for this package on your system and make it available on the classpath used by your server and client systems.

Here's the source code for the `MethodCounter` class, which implements the `CallCounterService`. Note that the Java class name does not have to be the same as the service name.

```
package javasoap.book.ch4;
public class MethodCounter {
   int _counter;
   public MethodCounter(  ) {
      _counter = 0;
   }
   public int getCount(  ) {
      return _counter;
   }
   public boolean doSomething(  ) {
      _counter += 1;
      return true;
   }
}
```

As you can see, there is nothing special about the Java code. `MethodCounter` contains a variable called `_counter` that keeps track of the number of method calls made during the lifetime of the object. The class contains a method called `doSomething( )` that increments the `_counter` variable and returns a Boolean value indicating whether the call was successful. There's also a method called `getCount( )` that returns the number of times `doSomething( )` has been called. We named the method `getCount( )` according to

the JavaBeans naming conventions for a read-only integer property named `Count`. We obey the JavaBeans conventions for a few reasons. First, it makes sense to follow standard conventions whenever we can. More important, we'll see later that some SOAP implementations can make use of JavaBeans properties without requiring the programmer to do extra work.

Note that our service class uses a no-argument constructor. This is important. Java SOAP implementations load service classes dynamically, usually by calling `Class.newInstance( )`. This method requires the class to have a constructor with no arguments. Of course, if we didn't explicitly include a constructor, a default no-argument constructor would have been created automatically. We include the constructor explicitly to highlight the fact that it's needed, regardless of how it is included.

Before we go any further, let's take a look at the SOAP envelope we would use to encapsulate the invocation of the `doSomething( )` method:

```
<SOAP-ENV:Envelope
    xmlns:SOAP-ENV=
       "http://schemas.xmlsoap.org/soap/envelope/"
    SOAP-ENV:encodingStyle=
       "http://schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
         <m:doSomething xmlns:m="CallCounterService">
         </m:doSomething>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The first child element of the SOAP `Body` is the method being invoked; the element is namespace-qualified using the service name (not the name of the class that implements the service). There are no parameters for the `doSomething` element, since there are no parameters to the `doSomething( )` method of our service class.

4.3 Deploying the Service

Service deployment is not part of the SOAP specification, so each implementation has its own deployment procedure. We'll look at service deployment using two SOAP implementations: Apache SOAP and GLUE.

4.3.1 Deploying with Apache SOAP

In Apache SOAP, you must create a *deployment descriptor*, which is an XML file that contains information about the service and the Java class that implements the service. Let's take a look at a deployment descriptor for `CallCounterService`, implemented by the Java class `javasoap.book.ch4.MethodCounter`:

```
<isd:service
   xmlns:isd="http://xml.apache.org/xml-soap/deployment"
   id="urn:CallCounterService">

   <isd:provider type="java"
         scope="Application"
         methods="doSomething getCount">
     <isd:java class="javasoap.book.ch4.MethodCounter"
         static="false"/>
   </isd:provider>
   <isd:faultListener>
   org.apache.soap.server.DOMFaultListener
   </isd:faultListener>
   <isd:mappings>
   </isd:mappings>
</isd:service>
```

The outermost XML element is called `service`, which is namespace-qualified using the identifier `isd`, representing the `http://xml.apache.org/xml-soap/deployment` namespace. The `service` element also has an `id` attribute that is assigned the unique name of the service being deployed. I've prefixed the service name, `CallCounterService`, with `urn`. You don't need to do this, but it's a common practice that I'll follow, at least in this example. Just be aware that it's not a requirement.

The first child element is called `provider` , which describes the implementation of the service. As with all the child elements of `service`, this element is also namespace-qualified with the `isd` identifier from the parent element. Three attributes are used on the `provider` element. The first attribute is `type` , which tells Apache SOAP what implementation type this service is using; we assign it the value `java`. (Apache SOAP allows for service implementations other than Java, but we won't be covering them.) The `scope` attribute specifies the service activation model of the deployed service. Possible values for this attribute are `Application`, `Session`, and `Request`. These correspond to the activation models described earlier. We're using `Application` level service activation now; we will experiment with other values later. The third attribute of the `provider` element is called `methods` . This attribute lists the method names that are exposed by the service. These method names have to match the corresponding methods in the Java class that implements the service because Apache SOAP uses dynamic class loading and Java reflection in order to make method calls. The method names are separated by spaces. For this example we're exposing the `doSomething` and `getCount` methods.

The `provider` contains a child element that further describes the details of the service implementation. We assigned the `type` attribute a value of `java`, which corresponds to the child element name `java`. The `java` element has two attributes and no data. The `class` attribute is assigned the full name of the Java class that implements the service, in this case `MethodCounter`. An attribute called `static` indicates whether the exposed methods of the service implementation class are static.

The next element of the deployment descriptor is a child element of `service` called `faultListener`. The data value of this element is the full name of the Java class used to create a fault listener object. This object is responsible for building SOAP faults to be returned when a fault condition exists. We're using

the `org.apache.soap.server.DOMFaultListener` class, which is provided by Apache SOAP. In Chapter 7 we'll look more closely at SOAP faults and write our own fault listener.

The last element is called `mappings` . In Chapter 6 we'll use this section to define Java classes that manage the serialization and deserialization of custom Java types. For now we don't need any mappings, so there are no entries in the mappings section.

Next, we need to use the deployment descriptor to deploy our service. I've named the deployment descriptor file *CallCounterService.dd*; it doesn't matter what name you use, as long as it makes sense and is recognizable to you. At this point, you should start your server, if it isn't running already. Apache SOAP includes a Java application that you can use from a command line to deploy a service. This application is called the service manager client (`org.apache.soap.server.ServiceManagerClient`); it can deploy, undeploy, and list services. Here's what the command to deploy the service looks like:

```
java org.apache.soap.server.ServiceManagerClient
  http://georgetown:8080/soap/servlet/rpcrouter deploy
CallCounterService.dd
```

The first parameter is the URL of the Apache SOAP RPC router; I'm running an Apache Tomcat server with Apache SOAP that accepts connections on port 8080. The server is a machine on the local network called `georgetown`. The second parameter, `deploy`, tells the service manager client that we are deploying a service. The last parameter, `CallCounterService.dd`, is the name of the file that contains the deployment descriptor. If all goes well, you won't see any output from the service manager client; you'll just return to the command prompt. If you get any exceptions or other error messages, look carefully at the output. Some likely problems are an incorrect classpath, or running the command from a directory other than the one that contains the deployment descriptor file.

After you've successfully deployed the service, you can verify it by calling the service manager client again. This time we'll ask the service manager to list all of the deployed services:

```
java org.apache.soap.server.ServiceManagerClient
  http://georgetown:8080/soap/servlet/rpcrouter list
```

The first parameter is the same as before: the URL of the Apache SOAP RPC router. `list` tells the service manager to list the currently deployed services. You should see the following response:

```
Deployed Services:
    urn:CallCounterService
```

Apache SOAP persists the information for the deployed services, so if you shut down the server that is hosting Apache SOAP and restart it at a later time, the services will still be available. It doesn't, however, persist the state of the objects that implement those services; they will start as if for the first time, with a new lifetime.

4.3.1.1 The Apache SOAP Admin tool

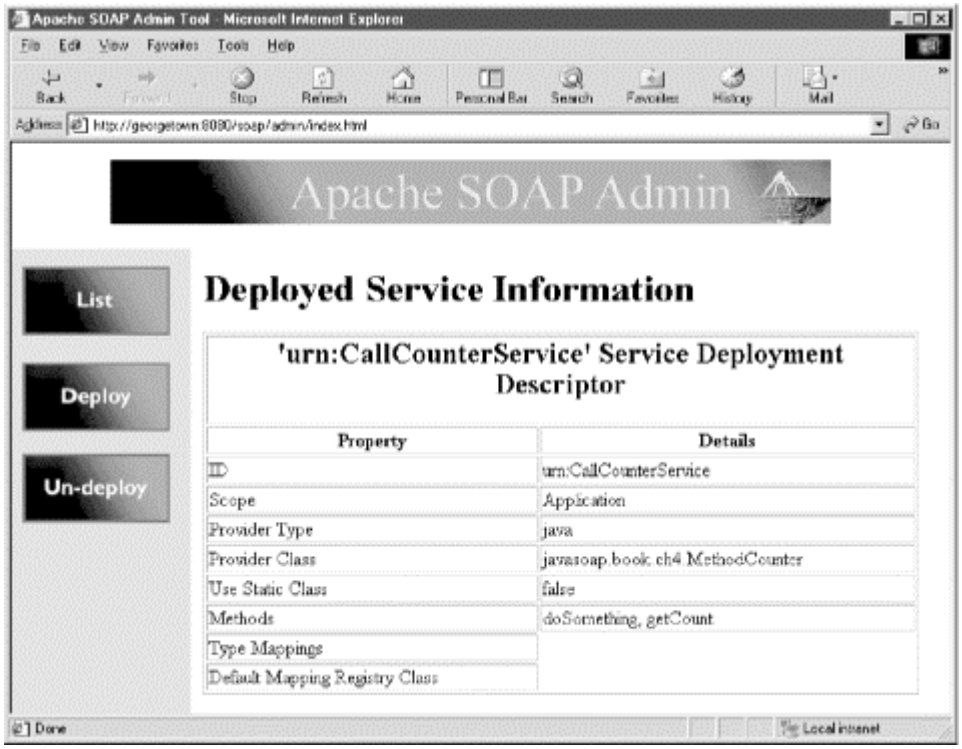
You might find it easier to use the web-based interface for Apache SOAP administration. The URL for this interface starts with the address and port of the server hosting Apache SOAP, followed by the SOAP Admin resource. I'm running Apache SOAP on `georgetown` on port 8080, so the URL is *http://georgetown:8080/soap/admin/index.html*. Figure 4-1 shows the main Apache SOAP administration page.

Figure 4-1. The Apache SOAP administration page

The List button takes you to a page that lists all of the deployed services. You'll see the `urn:CallCounterService` listed, as shown in Figure 4-2. The `urn:CallCounterService` link takes you to a page that shows the details of our service deployment, as shown in Figure 4-3. You can see all the deployment details that we specified in the deployment descriptor.

Figure 4-2. Currently deployed services

Figure 4-3. Service details

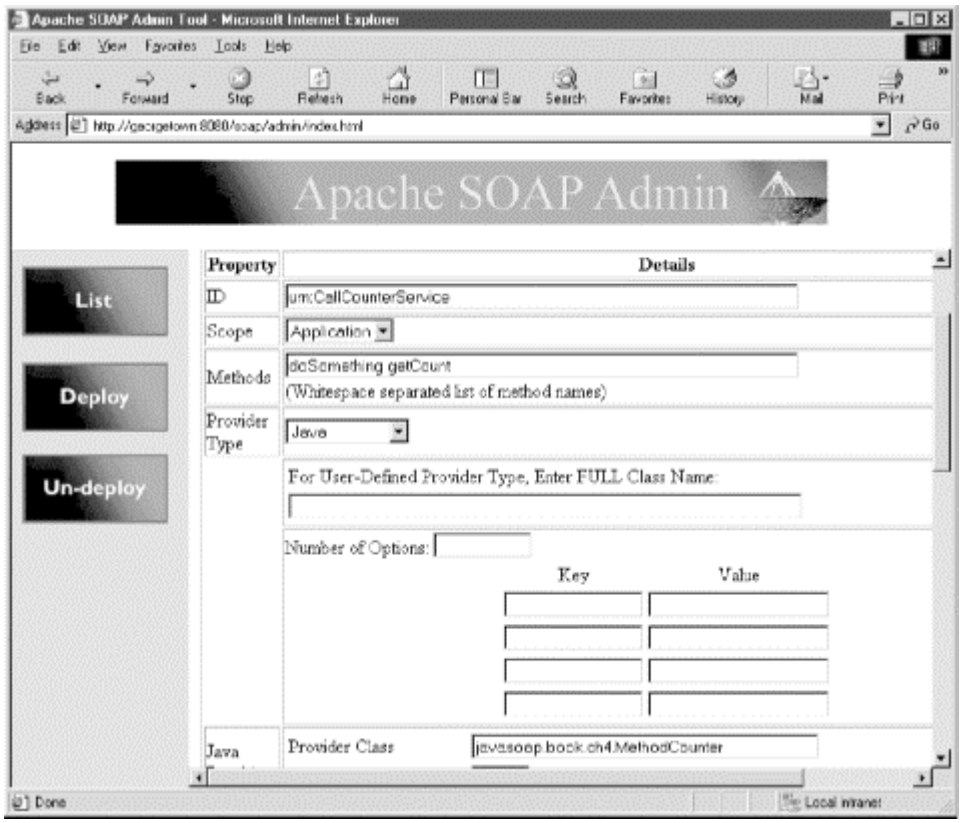


You can also deploy and undeploy services via this interface. If you click on Un-deploy, you'll find a list of deployed services. To undeploy a service, just click on its link. If you like, go ahead and undeploy the `urn:CallCounterService`. You can also use the service manager client from the command line to undeploy a service:

```
java org.apache.soap.server.ServiceManagerClient
  http://georgetown:8080/soap/servlet/rpcrouter undeploy
urn:CallCounterService
```

Now that the service is no longer deployed, click the Deploy button on the web interface. Figure 4-4 shows the form for service deployment. The information for deploying the service is already filled in.

Figure 4-4. Deploying a service with SOAP



To deploy the service, scroll down to the bottom of the form, where you'll find a button labeled Deploy. Click that button and the service will be deployed. You can use the web interface to look at the deployed service to make sure it looks the same as before. Apache SOAP makes no attempt to validate any of the information, so make sure you get it right.

There's an obvious security issue here. Would you want to allow anyone with the right URL to deploy or undeploy services? Not likely. Use the security features of the hosting technology in a production environment.

4.3.2 Deploying with GLUE

We can use the same Java class to implement `CallCounterService` in GLUE, because like Apache SOAP, GLUE does not require any special or extra code for services. However, the process of deploying GLUE services is quite different from that of Apache SOAP, and quite a bit simpler. GLUE provides a couple of ways to deploy services. We'll deploy the service by writing a standalone application that starts an instance of GLUE's integrated HTTP server within our own code, and then use the GLUE classes and related methods to publish the service. Here's the application:

```
package javasoap.book.ch4;
import electric.util.Context;
import electric.registry.Registry;
import electric.server.http.HTTP;
public class CallCounterApp {
   public static void main( String[] args )
      throws Exception {

      HTTP.startup("http://georgetown:8004/glue");
      Context context = new Context(  );
      context.addProperty("activation", "application");
      Registry.publish( "urn:CallCounterService",
         javasoap.book.ch4.MethodCounter.class, context );
   }
}
```

The `CallCounterApp` class contains the required `static main( )` method so it can be called from the command line. The `HTTP.startup( )` method starts an instance of an HTTP server on my local machine `georgetown` on port 8004, and the resource used to accept and route SOAP messages is `/glue`. Putting that together gives us the full URL of *http://georgetown:8004/glue*, which is passed as a parameter to `HTTP.startup( )`.

After starting the server, we need to register our service with it. We start by specifying the activation model we want to use. To do so, we create an instance of the class `electric.util.Context`. This class is part of the GLUE APIs, and is used to assemble a collection of property names and associated values. To record the activation model, we call the `addProperty( )` method of the `context` object, giving it the property name `activation` and the associated value `application`. Finally, we're ready to publish the service, using the `publish( )` method of GLUE's `electric.registry.Registry` class. The first parameter passed is `urn:CallCounterClass`, which once again is the name of the service being published. The second parameter is the Java class that implements the service, `MethodCounter`. The `context` object is passed as the last parameter, indicating that we want to publish (deploy) the service using the application-level service activation model.

To start the application, we enter the following from a command-line prompt:

```
java javasoap.book.ch4.CallCounterApp
```

When we start the application, we'll see some output telling us the GLUE version number as well as the full URL of the GLUE resource that is listening for SOAP messages. This is the same URL that was passed to the `HTTP.startup( )` method. At this point `the urn:CallCounterService` service is deployed and ready to accept SOAP RPC method invocations.

How does GLUE know what methods are being exposed as part of the service? In the Apache SOAP deployment descriptor we specifically defined the methods that were exposed; we haven't done anything like that for GLUE. Instead, GLUE uses a number of techniques to determine the methods automatically. Therefore, the simplest way to deploy a service is to let GLUE figure out which methods to expose on its own. In later chapters we'll look at how to gain control over the methods that are exposed. For this example, GLUE exposes all the public methods of the `MethodCounter` class except for the constructor. You shouldn't be surprised that GLUE requires a no-argument constructor, just like Apache SOAP.

4.3.2.1 The GLUE console

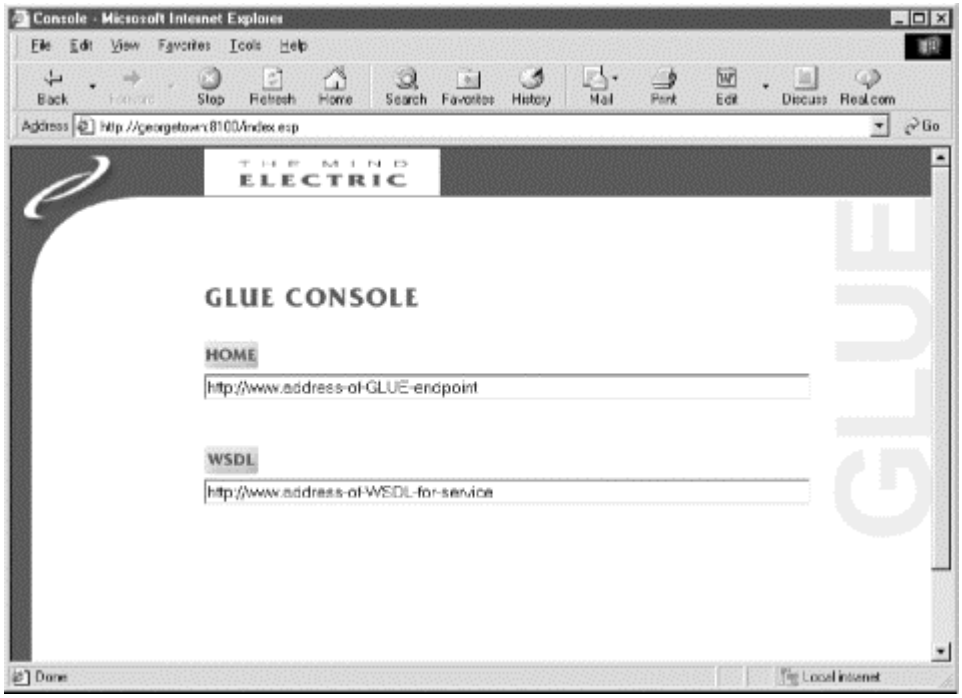
GLUE includes a web-based interface that allows you to see the available services on a given server, edit them, and even deploy new services. The GLUE console is a special web application that can be started using the *console.bat* (or *console.sh*) script that is included in the */bin* directory of the GLUE installation. If you're using a command-line prompt in a Microsoft Windows environment, I suggest you navigate to that directory to execute the `console` command, because `console` is a Windows command as well. If you navigate to GLUE's */bin* directory first, you'll start the correct program.

By default, the GLUE console listens for HTTP connections to the local machine on port 8100 (i.e., *http://localhost:8100*). To run the console application on another port, specify that port number as an argument to the `console` command. The following command starts the console on port 8150:

```
console 8150
```

All the examples in this book use the default port of 8100. So go ahead and start the console application, then launch your web browser and go to *http://localhost:8100*. You will see the main GLUE console window shown in Figure 4-5.

Figure 4-5. The GLUE console



For now, we're going to use only the first field of the GLUE console interface, labeled HOME. Remember that we have already started an instance of a GLUE server using the URL *http://georgetown:8004/glue*. Enter this URL in the HOME field of the console form, click on the HOME button, and you'll be taken to the page shown in Figure 4-6. This page includes the URL of the server's endpoint, the resource that manages the SOAP messages. It also shows all of the services currently available on that server. The first one, `system/admin`, is a special service used by GLUE. The second service is `urn:CallCounterService`, which we deployed

earlier. The `urn:CallCounterService` link takes you to a page that describes the web service, shown in Figure 4-7.

Figure 4-6. The GLUE home

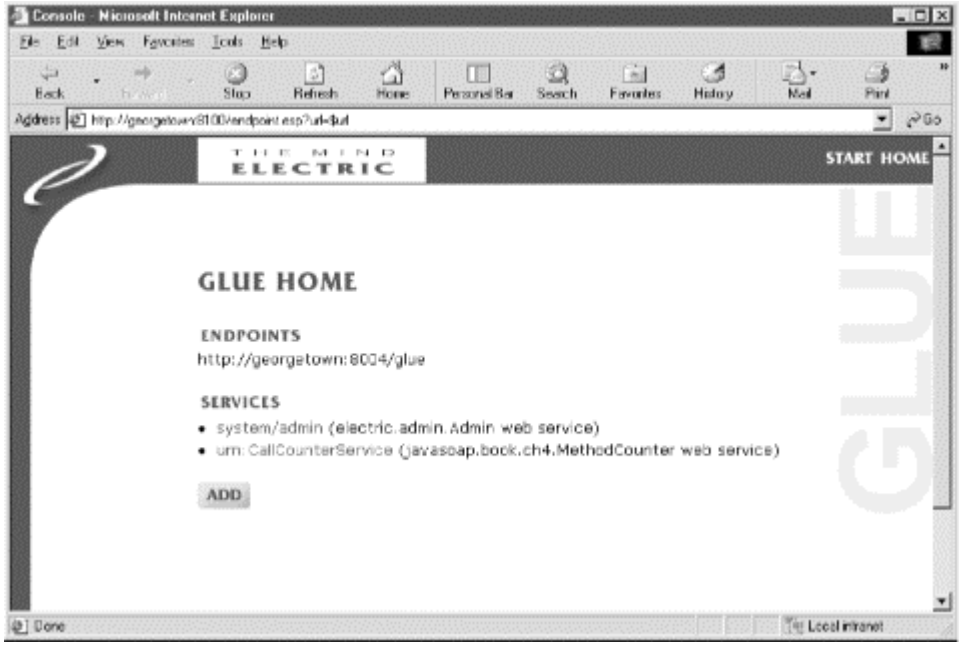


Figure 4-7. Web service information

Let's take a look at the information provided for a web service. The Description field provides a description of the service; its value is a default value that GLUE has generated for us. In later examples we'll provide a more useful description. The Endpoint field contains the full URL of the service. The WSDL field shows a URL that can be used to request a description of the service in the Web Service Description Language; we'll look at this in Chapter 9. The Class field shows the full class name of the Java class that implements the service, and the Mode field shows that the service activation model used by this service is `application`.

Finally, the Methods field contains the methods that can be invoked on this service. These were automatically determined by GLUE.

The EDIT button takes you to a form that allows you to edit the description, Java class, and mode (service activation model) of the service. It also allows you to modify the URN (universal resource name). The URN is the actual name of the service, which in this case is `urn:CallCounterService`. Clicking the JAVA button gives you a listing of client-side Java classes that can be used to access the service. We'll get to this in Section 4.4.2.

You can undeploy (delete) the service by clicking the DEL button. If we go ahead and delete the service, the GLUE console responds with the page shown in Figure 4-8.

Figure 4-8. Successful deletion

At this point, click the "here" link to go back to the page that describes the available web services on the server. This will look similar to Figure 4-6 except that `urn:CallCounterService` will no longer be listed. Now click the ADD button, which takes you to a form for providing the information needed to deploy a service (Figure 4-9). I've entered the information for the `urn:CallCounterService`, this time providing a better description. I used the same URN, class, and mode used when we deployed the service from within the Java application.

Figure 4-9. Adding a service through the console

Press the SAVE button to deploy the service. If you provide an invalid class name, the GLUE console will indicate a failure. Otherwise you'll be taken to the web service page (Figure 4-7).

4.4 Writing Service Clients

In the next few sections, we'll write some Java code that invokes methods on the services we've exposed in both Apache SOAP and GLUE. For now, we won't mix and match technologies — we'll use Apache SOAP APIs to call an Apache SOAP server, and GLUE APIs to call a GLUE server.

So what about interoperability? One of the most important aspects of SOAP as a wire protocol is that your choice of implementation should not prohibit you from communicating successfully with other SOAP implementations. However, there are still some problems with SOAP interoperability. Some technologies are better at it right now than others, and in some cases you have to jump through a few hoops to make different SOAP technologies communicate properly with each other. So let's put off this subject until Chapter 9, where we'll cover these issues in detail.

4.4.1 Calling the Service with Apache SOAP

Now that we have a service up and running, let's write some Java code that acts as a client of the service. Here's a simple Java application that invokes the `getCount` method of the `urn:CallCounterService` on our Apache SOAP server:

```
package javasoap.book.ch4;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class GetCountApp {
   public static void main(String[] args)
      throws Exception {

      URL url =
        new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call(  );
      call.setTargetObjectURI("urn:CallCounterService");
      call.setMethodName("getCount");
      try {
         Response resp = call.invoke(url, "");
         Parameter ret = resp.getReturnValue(  );
         Object value = ret.getValue(  );
         System.out.println("Result is " + value);
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                         e.getFaultCode(  ) + "): " +
                         e.getMessage(  ));
      }
   }
}
```

This program starts by creating an instance of `java.net.URL` using the URL of the Apache SOAP RPC router of our Apache SOAP server. It then creates an instance of `org.apache.soap.rpc.Call`, a Java class that encapsulates a SOAP RPC method call. The `setTargetObjectURI( )` method of the `call` object is used to specify the name of the service, and `setMethodName( )` specifies the name of the method being invoked.

At this point we're ready to invoke the method on the specified service. Since the method invocation can throw a `SOAPException`, we put it within a `try/catch` block. If anything goes wrong we'll print out the fault code and message from the exception.

To make the method call, we use the `invoke( )` method of the `call` object, with the URL that we set up earlier as the first parameter. The second parameter is a SOAP action URI, which is included as an HTTP header entry. We're not using a SOAP action, so an empty string is passed for this parameter. The `invoke( )` method returns an instance of `org.apache.soap.rpc.Response`; upon successful return, this object contains the response from the server. For now let's assume that the call succeeds. After invoking the method, we call the `getReturnValue( )` method of the response object. This method returns an instance of `org.apache.soap.rpc.Parameter`, which contains the return value, or response, of the service method call. Now we call the `getValue( )` method of the parameter, which returns an instance of `Object`; the actual type of the object you get back depends on the return type of the method. This is an important point. Apache SOAP RPC method calls return instances of Java objects, not instances of Java primitives. In a case like this, where the service method being called returns an `int` value, the primitive return value is created as an instance of its corresponding Java object type — in this case, `Integer`. We could cast the return value to `Integer`, but instead we'll leave it as an `Object` instance and print the result using

`System.out.println( )`. This method implicitly calls `toString( )`, which returns the `Integer`'s value as a `String`.

Let's go ahead and run this application:

```
java javasoap.book.ch4.GetCountApp
```

If everything is set up correctly, you'll see the following output:

```
Result is 0
```

The result is 0 because we haven't yet made any calls to the `doSomething( )` method that increments the value of the counter in the service object. This is great! We've successfully deployed and invoked a SOAP service in Java. Let's take a look at the SOAP payload that was used to invoke the service:

```
POST /soap/servlet/rpcrouter HTTP/1.0
Host: Georgetown
Content-Type: text/xml; charset=utf-8
Content-Length: 344
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<SOAP-ENV:Body>
  <ns1:getCount xmlns:ns1="urn:CallCounterService">
</ns1:getCount>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

All of the data in this SOAP document should be familiar to you. The information specific to this example is contained in the SOAP body, where you can see that the `getCount` element (which represents the method being invoked) is namespace-qualified by `urn:CallCounterService`, the name of the service on which the method is being invoked. While we're at it, let's look at the response sent back by the server:

```
HTTP/1.0 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: 468
Set-Cookie2: JSESSIONID=yqf5f6su31;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=yqf5f6su31;Path=/soap
Servlet-Engine: Tomcat Web Server/3.2.3

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
<SOAP-ENV:Body>
 <ns1:getCountResponse
  xmlns:ns1="urn:CallCounterService"
  SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/">
   <return xsi:type="xsd:int">0</return>
 </ns1:getCountResponse>

</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The HTTP headers are shown so that you can see the entire response. Notice that the server is returning HTTP cookie values. Although we aren't using them now, those cookies will be useful when we use `Session` service activation for services.

Now look at the SOAP body. Since the name of the method that was invoked is `getCount`, the first child element of the body is `getCountResponse`, namespace-qualified by the service name. The `SOAP-ENV:encodingStyle` attribute is set to `http://schemas.xmlsoap.org/soap/encoding`. Within the `getCountResponse` element we find a single child element named `return`, which contains the return value of the `getCount` method. `return` is explicitly typed as `xsd:int`, and contains the data value `0`.

Let's make a few calls to `doSomething` in order to bump up the counter:

```
package javasoap.book.ch4;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class GetCountApp2 {
   public static void main(String[] args)
      throws Exception {

      URL url = new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call(  );
      call.setTargetObjectURI("urn:CallCounterService");
      try {
         call.setMethodName("doSomething");
         Response resp = call.invoke(url, "");
         resp = call.invoke(url, "");
         resp = call.invoke(url, "");
         call.setMethodName("getCount");
         resp = call.invoke(url, "");
         Parameter ret = resp.getReturnValue(  );
         Object value = ret.getValue(  );
         System.out.println("Result is " + value);
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                         e.getFaultCode(  ) + "): " +
                         e.getMessage(  ));
      }
   }
}
```

In this example, we first set the name of the method to `doSomething` using `setMethodName( )`, and then we execute `call.invoke( )` three times. Then the method name is set to `getCount` and the service is invoked again. Like last time, the result of the `getCount` method is displayed. You should see the following output after running this application from a command-line prompt:

```
Result is 3
```

We've deployed `urn:CallCounterService` using application scope, meaning that a single instance of the service class is used for every service request. So each time the `doSomething` method is called, the `_counter` variable inside the `MethodCounter` instance is incremented. If you run the application again, the result will be 6. One more time and it'll be 9. Here you see application scope services in action, trivial as the example itself happens to be. If you want to get the counter back to 0, shut down the server you're using to host Apache SOAP and restart it.

You'd think that an alternative to shutting down the server would be to undeploy and then redeploy the service. It doesn't work, though, at least not with a Tomcat server. Apparently the implementing service class is cached and reused if the service is redeployed. If you're using some other server to host Apache SOAP, you should check this out for yourself.

4.4.2 Calling the Service with GLUE

Now let's create an application that uses the GLUE APIs to communicate with the GLUE server we started earlier. GLUE takes a different approach to Java client programming. It allows you to access services via local Java interfaces. These interfaces contain methods that correspond to the exposed, or published, methods of the service. In this example, we want to write a Java interface that has the `getCount( )` and `doSomething( )` methods just as they appear in `MethodCounter`. Here's such an interface, named `ICallCounterService`.[5]

```
package javasoap.book.ch4;
public interface ICallCounterService {
  int getCount(  );
  boolean doSomething(  );
}
```

Next let's create an application that accesses the GLUE-hosted service `urn:CallCounterService`.

[5] Starting interface names with an I is a common, but by no means universal, convention.

```
package javasoap.book.ch4;
import electric.registry.Registry;
import electric.registry.RegistryException;
public class GetCountApp3 {
   public static void main(String[] args) throws Exception
   {
      try {
         ICallCounterService ctr =
           (ICallCounterService)Registry.bind(
             "http://georgetown:8004/glue/urn:CallCounterService.wsdl",
              ICallCounterService.class);
       System.out.println("Result is " + ctr.getCount( ));
    }
    catch (RegistryException e)
    {
       System.out.println(e);
    }
  }
}
```

GLUE allows us to bind the service interface to the service. Once this is accomplished, we simply use the interface as if it were a local object (which it is!). So the only step is to call the static `bind( )` method of the `electric.registry.Registry` class. The first parameter is a URL to the WSDL file that describes the service. GLUE is based on WSDL, and automatically generates the WSDL description for the service. (WSDL is covered briefly in Chapter 9.) To create the proper URL, start with the address of the server and resource (*http://georgetown:8004/glue*) and add the name of the service, followed by *.wsdl*. The second parameter is the Java class that represents the interface, `ICallCounterService.class`. The return value of the `Registry.bind( )` method is cast to an instance of the Java interface for the service, `ICallCounterService`.

After the call to `Registry.bind( )` returns, we can use the `ctr` variable to call the `getCount( )` and `doSomething( )` methods as if it were nothing more than a local Java object. In this example we simply print out the result by calling `ctr.getCount( )`. Here's how to run the example:

```
java javasoap.book.ch4.GetCountApp3
```

The result of calling `getCount( )` is 0 because there have not yet been any calls to `doSomething( )`:

```
Result is 0
```

The GLUE console can generate some Java code that makes it easier to access the service. To generate this code, go back to the GLUE console using your browser, then go to the `urn:CallCounterService` and click on the JAVA button. GLUE uses a different name for the interface (`IMethodCounter`) than I did, but the contents of the interface are otherwise the same. The helper class is called `MethodCounterHelper`. It encapsulates the `Registry.bind( )` calls and the cast required to pass the correct interface. The helper class provides two static `bind( )` methods. The first one takes no parameters and generates the URL of the service endpoint automatically. The other `bind( )` allows you to specify another URL if you like. Here's the previous client application modified to use the Java interface and helper class generated by the console. Make sure that you use the appropriate

package name for the interface and helper class. The GLUE console does not do that part, so it's up to you.[6]

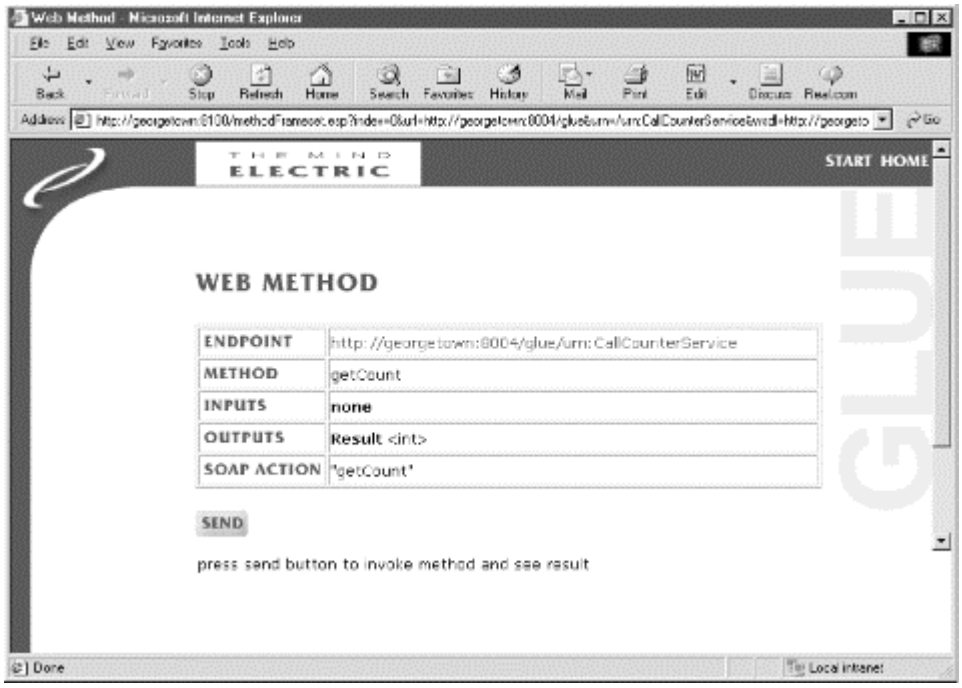
```
package javasoap.book.ch4;
import electric.registry.RegistryException;
public class GetCountApp4 {
   public static void main(String[] args) throws Exception
   {
      try {
       IMethodCounter ctr = MethodCounterHelper.bind(  );
       System.out.println("Result is " + ctr.getCount(  ));
    }
    catch (RegistryException e)
    {
       System.out.println(e);
    }
  }
}
```

Using the helper class `MethodCounterHelper` cleans up the code a bit. For a small program like this, it's not a significant difference, but there would be a bigger payoff in a larger application. Compile the new client, together with the Java interface and helper class as part of the `javasoap.book.ch4` package. Running this example from a command-line prompt gives us the same result as before.

The GLUE console also provides a way to invoke the methods of a service without writing any code. Go back to the `urn:CallCounterService` web service page in your browser. The methods that are listed in the METHODS field are hyperlinks; clicking on one of these links brings you to a form for invoking the selected service method. If you select the `getCount` method, you'll see the form shown in Figure 4-10. There are fields for the input parameters and return value, labeled INPUTS and OUTPUTS. If there were any parameters for the `getCount` method, there would be a place to enter a value. In this case there are none. Clicking on the SEND button invokes the service method. When the call returns, the result is displayed below the SEND button; if you do this now, you'll see a 0. This web interface provides a useful tool for checking the functionality of web services without having to write Java code.

[6] GLUE provides a utility called `wsdl2java` that generates Java code from a WSDL file, and has an option to generate the code for a specified package. We'll look at this utility in Chapter 9.

Figure 4-10. Invoking getCount through the browser



Let's modify the GLUE example to make multiple calls to `doSomething` before calling `getCount`. Here's a new version of our client that uses the GLUE APIs to make multiple calls, using the interface and helper class generated by the GLUE console:

```
package javasoap.book.ch4;
import electric.registry.RegistryException;
public class DoSomethingApp {
   public static void main(String[] args) throws Exception
   {
      try {
       IMethodCounter ctr = MethodCounterHelper.bind(  );
       ctr.doSomething(  );
       ctr.doSomething(  );
       ctr.doSomething(  );
       System.out.println("Result is " + ctr.getCount(  ));
    }
    catch (RegistryException e)
    {
       System.out.println(e);
    }
  }
}
```

When you run this client for the first time, you get the following output:

```
Result is 3
```

If you run it again, the result will be 6, and so on, because the service is using application scope (activation mode). To reset the counter to 0, stop and then restart the `CallCounterApp` application, or use the GLUE console to delete the service and then add it back. With the GLUE console, the latter approach works just fine.

4.5 Deploying with Request-Level Scope

Up until now, all the examples we've created have been deployed using application-level scope (service activation). In this mode, a single instance of the implementing service class handles all of the method invocations. Let's change the deployment to request-level scope. With request-level scope, a new instance of the service object is created for each method invocation, and that object is destroyed when the invocation is complete.

First we'll change the example running under Apache SOAP by modifying its deployment descriptor. The only change required is to set the `scope` attribute of the `provider` element to `Request`; the rest of the deployment descriptor remains exactly the same:

```
<isd:service
   xmlns:isd="http://xml.apache.org/xml-soap/deployment"
   id="urn:CallCounterService">

   <isd:provider type="java"
         scope="Request"
         methods="doSomething getCount">
     <isd:java class="javasoap.book.ch4.MethodCounter"
         static="false"/>
   </isd:provider>
   <isd:faultListener>
   org.apache.soap.server.DOMFaultListener
   </isd:faultListener>
   <isd:mappings>
   </isd:mappings>
</isd:service>
```

Remember to undeploy and then redeploy the `urn:CallCounterService` using any of the techniques discussed earlier. You don't have to shut down the server the way we did before; now that we're using request-level scope, there shouldn't be any instance of `MethodCounter` in memory anyway. So you can use the command-line version of the service manager client to undeploy and redeploy the service, or you can use the Apache SOAP Admin browser application to do the same. If you decide to shut down the server and restart it, you'll still have to undeploy and redeploy the service because Apache SOAP maintains deployed services persistently.

Now that you've redeployed `urn:CallCounterService` using request-level scope, you can use the Apache version of the client application to access the service. Even though the application calls `doSomething( )` multiple times before calling `getCount( )`, the result will always be 0. This is a direct consequence of request-level scope: whenever either of these methods is invoked, a new instance of the `MethodCounter` class is created, and that instance is destroyed when the invocation is complete. No state is maintained across invocations. Therefore, there's no way for the count to be anything but 0.

Let's do the same for the service running under GLUE. The easiest way to accomplish this is to use the GLUE console. Browse to the web services page for the `urn:CallCounterService` service, and then click the EDIT button. This takes you to the web service editor page shown in Figure 4-11.

Figure 4-11. The web service editor

To change the activation mode, click the corresponding radio button in the MODE field and select Request. Now click the SAVE button, and the service is redeployed using request-level scope.

I've been using different terms for the service activation model. Sometimes I call it scope, sometimes activation mode, and other times service activation model. They all mean the same thing, but different implementations use different terms and I want you to get used to all of them. The words may be different, but they all still refer to the lifetime of the object that implements the service.

You can check that the service is now using request-level scope by running the `DoSomethingApp` application again. You will always get a result of 0. Again, the calls to `doSomething` and the call to `getCount` are all made against separate instances of the `MethodCounter` class.

4.6 Deploying with Session-Level Scope

Apache SOAP provides for session management by passing cookies via the HTTP headers. Earlier in this chapter, we saw an HTTP response from the server that included HTTP header entries called `Set-Cookie` and `Set-Cookie2`. The client application uses these cookies if it needs to make subsequent method invocations against the same session as the original request. The Apache SOAP API uses a simple technique for handling this. `org.apache.soap.transport.http.SOAPHTTPConnection` includes a method called `setMaintainSession( )` that takes a single `boolean` parameter. This parameter, when set to `true`, tells the connection object to maintain the current session. The connection object implements this by keeping track of the cookies and sending them back to the server when the next method invocation takes place.

Go ahead and edit the deployment descriptor, setting the `scope` attribute to `Session`. Now run the client program again. You'll get the following output:

```
Result is 3
```

This output proves that the same instance of the service object was used for all of the method invocations. If you run it again you'll get the same result because a new instance of the session object is created on the first call to `doSomething( )`, and each subsequent method call uses the same service object. Because we're using the same instance of `org.apache.soap.rpc.Call` for every service call, we're also using the same instance of `org.apache.soap.transport.http.SOAPHTTPConnection`. You've probably guessed that the default behavior of `org.apache.soap.transport.http.SOAPHTTPConnection` is to maintain the session.

A common use for session-level scope is a shopping cart, where you want to keep track of the session between service method invocations. However, there may be times when you do not want to use the same session for all of your method calls, even though the service is deployed using session scope. In the next example, the `SOAPHTTPConnection` object is retrieved from the `Call` object after the first method invocation. Then a call to the connection's `setMaintainSession( )` method is made with a parameter of `false`. This stops the connection from using the same session on subsequent service method calls.

```
package javasoap.book.ch4;
import java.net.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import org.apache.soap.transport.http.*;
public class SessionApp {
   public static void main(String[] args)
      throws Exception {

      URL url = new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call(  );
      call.setTargetObjectURI("urn:CallCounterService");
      try {
         call.setMethodName("doSomething");
         Response resp = call.invoke(url, "");
         SOAPHTTPConnection st =
          (SOAPHTTPConnection)call.getSOAPTransport(  );
         st.setMaintainSession(false);
         resp = call.invoke(url, "");
         st.setMaintainSession(true);
         resp = call.invoke(url, "");
         call.setMethodName("getCount");
         resp = call.invoke(url, "");
         Parameter ret = resp.getReturnValue(  );
         Object value = ret.getValue(  );
         System.out.println("Result is " + value);
      }
```

```
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                            e.getFaultCode( ) + "): " +
                            e.getMessage( ));
      }
   }
}
```

Before the third call to `doSomething( )`, we call `setMaintainSession( )` again with a parameter of `true`. Therefore, after the third call to `doSomething( )`, the `SOAPHTTPConnection` object will keep track of the session cookies. Then, when we call `getCount( )`, those cookies will be sent back to the server, ensuring that the same session object is used again. You should get the following output when you run the `SessionApp` application:

```
Result is 1
```

If you want to maintain a single session across multiple calls, make sure that you establish the session before the first call that should be part of that session. However, programming this way is a little sloppy, at least for my taste. I would prefer to create a new instance of the `SOAPHTTPConnection` and use it when I begin making calls using a single session. Of course, these considerations apply only to services deployed using session scope. If you'd like to give this a try, change the code within the `try` block to look like this:

```
call.setMethodName("doSomething");
Response resp = call.invoke(url, "");
resp = call.invoke(url, "");
SOAPHTTPConnection st = new SOAPHTTPConnection( );
call.setSOAPTransport(st);
resp = call.invoke(url, "");
call.setMethodName("getCount");
resp = call.invoke(url, "");
Parameter ret = resp.getReturnValue( );
Object value = ret.getValue( );
System.out.println("Result is " + value);
```

The lifetime of a server session is implementation dependent. The server platform you are using, the way it's configured, and the available configuration options all play a part in how long a session lasts. In most cases, the server terminates a session after some period of idle time has passed. GLUE manages sessions automatically, and lets you set the session lifetime by calling `electric.server.http.HTTP.setSessionTimeout( )`. The parameter passed to this method is an integer that contains the number of seconds that the session can remain inactive before it is invalidated.

4.7 Passing Parameters

So far, the examples in this chapter have used methods that don't contain any parameters. We'll now spend a little time looking at how parameters are passed. Let's create a service called `urn:StockPriceService`, implemented by the `javasoap.book.ch4.StockPrice` class. The `getPrice( )` method takes parameters for the stock symbol as well as the currency. The method always returns the float value of 75.33 — not a very interesting stock, but at least it won't go down. Seriously though, we're just interested in how parameters are

passed. The details of accessing a database or data feed to find a stock price are, for the moment, left to you.

```
package javasoap.book.ch4;
public class StockPrice {
   public float getPrice(String stock, String currency) {
      float result;
      // determine the price for stock and return it
      // in the specified currency
      result = (float)75.33;
      return result;
   }
}
```

We'll gloss over the deployment details, both for Apache SOAP and GLUE, since we've spent much of this chapter on that subject. Here's the Apache SOAP deployment descriptor for the `StockPriceService` implemented by the `StockPrice` class. The service uses application-level scope, and contains a single method called `getPrice` that retrieves the current trading price of a stock using a specified currency.

```
<isd:service
   xmlns:isd="http://xml.apache.org/xml-soap/deployment"
   id="urn:StockPriceService">

   <isd:provider type="java"
         scope="Application"
         methods="getPrice">
     <isd:java class="javasoap.book.ch4.StockPrice"
         static="false"/>
   </isd:provider>
   <isd:faultListener>
   org.apache.soap.server.DOMFaultListener
   </isd:faultListener>
   <isd:mappings>
   </isd:mappings>
</isd:service>
```

This service can be deployed in Apache SOAP using the browser-based Admin tool or by putting this deployment descriptor into a file and using the service manager client. We'll use the latter approach. The file containing the deployment descriptor is *StockPriceService.dd*. Using that file, the following command will deploy the service on my local machine `georgetown`:

```
java org.apache.soap.server.ServiceManagerClient
  http://georgetown:8080/soap/servlet/rpcrouter deploy StockPriceService.dd
```

Here is a simple application using the Apache SOAP APIs to access the `StockPriceService`:

```
package javasoap.book.ch4;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
public class StockPriceApp {
   public static void main(String[] args)
      throws Exception {

      URL url = new URL(
          "http://georgetown:8080/soap/servlet/rpcrouter");

      Call call = new Call(  );
      call.setTargetObjectURI("urn:StockPriceService");

      call.setEncodingStyleURI(Constants.NS_URI_SOAP_ENC);
      String stock = "MINDSTRM";
      String currency = "USD";
      Vector params = new Vector(  );
      params.addElement(new Parameter("stock",
                        String.class, stock, null));
      params.addElement(new Parameter("currency",
                        String.class, currency, null));
      call.setParams(params);

      try {
         call.setMethodName("getPrice");
         Response resp = call.invoke(url, "");
         Parameter ret = resp.getReturnValue(  );
         Object value = ret.getValue(  );
         System.out.println("Price is " + value);
      }
      catch (SOAPException e) {
         System.err.println("Caught SOAPException (" +
                        e.getFaultCode(  ) + "): " +
                        e.getMessage(  ));
      }
   }
}
```

The beginning of this application is similar to the applications we created earlier using the Apache SOAP APIs. The code for setting up parameters starts with a call to the `setEncodingStyleURI( )` method of the `Call` object. The parameter passed to this method is a string that contains the URI of the encoding style that should be used to serialize the parameters. In this case, the encoding style is given by `Constants.NS_URI_SOAP_ENC`, which is a constant defined by the `org.apache.soap.Constants` class. The value of this constant is `"http://schemas.xmlsoap.org/soap/encoding/"`, which is the standard SOAP encoding namespace. The default value for the encoding style of a `Call` instance is `null`, so you must remember to call the `setEncodingStyleURI( )` method. Without this, the server side will not understand the encoding scheme that should be used to deserialize the parameters.

Next, we set up two `String` variables that contain the stock symbol[7] and the currency to use when the `getPrice( )` method is invoked. Now we create an instance of `java.util.Vector`, which will hold the parameters. Each parameter is held in an instance of `org.apache.soap.rpc.Parameter`. The constructor of this class takes the parameter's name,

[7] MINDSTRM is not a real stock symbol.

the class of the object that contains the parameter value, the object that contains the parameter value, and the encoding style URI for the parameter. For the first parameter of the `org.apache.soap.rpc.Parameter` constructor we pass the name `stock`, which is the name used for the first parameter of the `getPrice` method of the `StockPrice` class. The second parameter is `String.class`, which tells the constructor that the value is passed as a Java `String` instance. The third parameter is a `String` variable that contains the actual value we're passing: `"MINDSTRM"`. We pass `null` for the last parameter, indicating that an encoding style URI should not be included for the XML element that will be serialized for this parameter. Remember that we've already set up the encoding style URI for the entire call when we called `call.setEncodingStyleURI( )`; unless you want to use some other encoding style for this particular parameter, just pass `null`. The `org.apache.soap.rpc.Parameter` instance is added to the vector using the `params.addElement( )` method. We go through the same process for the `currency` parameter, adding it to the vector, `params`, as well.

Finally, we pass the vector containing the parameters to the `Call` object using the `call.setParams( )` method. From here, the rest of the code is familiar. The `getPrice( )` service method is invoked, and the result is printed for display. Here's the SOAP envelope that is sent to the server:

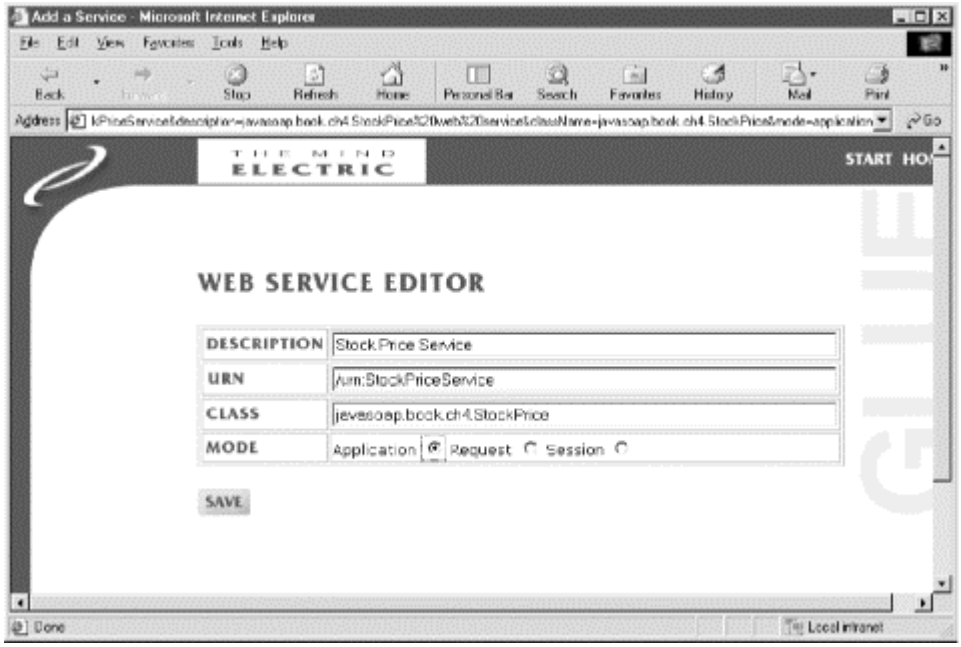
```
<SOAP-ENV:Envelope
 xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<SOAP-ENV:Body>
  <ns1:getPrice
     xmlns:ns1="urn:StockPriceService"
     SOAP-ENV:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/">
     <stock xsi:type="xsd:string">MINDSTRM</stock>
     <currency xsi:type="xsd:string">USD</currency>
  </ns1:getPrice>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

As we've seen before, the name of the method we're calling is used for the child element of the SOAP `Body`, namespace-qualified by the name of the service itself. The `SOAP-ENV:encodingStyle` attribute on the `getPrice` element applies to all of its subelements, unless those elements include the `SOAP-ENV:encodingStyle` attribute themselves. (That's why we passed `null` to the `org.apache.soap.rpc.Parameter` constructor.) The two parameters are serialized as child elements of `getPrice`. Each one is explicitly typed using the `xsi:type` attribute with a value of `xsd:string`, and the parameter value is serialized as the element data value for the associated parameter.

Now let's take a look at what's required to pass parameters using the GLUE APIs. We've already seen that GLUE binds Java interfaces to services, and the methods of those interfaces are treated like any other Java instance. So there's really nothing profound about passing parameters in GLUE-based applications. But let's go through the steps anyway, just to reinforce the concepts.

Assuming that the `CallCounterApp` application is still running, we can use the GLUE console to deploy the new service on that GLUE server instance. Clicking the ADD button on the home page of your GLUE server brings you to an empty web service editor form. Fill in the appropriate service details, as shown in Figure 4-12.

Figure 4-12. Specifying details with the web service editor



In the figure, I've given the service a short description, as well as the service's URN (`urn:StockPriceService`). The Java class implementing the service is `StockPrice`, and the service activation mode is Application. Click SAVE to deploy the service.

Now we need a Java interface to bind to this service. You can use the one generated by the GLUE console, if you prefer; for this example, we'll use the interface `IStockPriceService`, listed below. In either case, remember that the names used by the GLUE console when it generates an interface are different from the names we use when creating the interfaces by hand.

```
package javasoap.book.ch4;
public interface IStockPriceService {
  float getPrice(String stock, String currency);
}
```

Writing an application that accesses the `StockPriceService` using the GLUE APIs is simple. We won't use the helper class by the GLUE console here, although there's certainly no reason why you couldn't. Here's the application for accessing the service:

```
package javasoap.book.ch4;
import electric.registry.Registry;
import electric.registry.RegistryException;
public class StockPriceApp2 {
   public static void main(String[] args) throws Exception
   {
      try {
         IStockPriceService ctr =
           (IStockPriceService)Registry.bind(
              "http://georgetown:8004/glue/urn:StockPriceService.wsdl",
              IStockPriceService.class);
         String stock = "MINDSTRM";
         String currency = "USD";
         System.out.println("Price is " +
           ctr.getPrice(stock, currency));
      }
      catch (RegistryException e)
      {
        System.out.println(e);
      }
   }
}
```

Once you bind an interface to the service, you work directly with the Java interfaces as if the service objects were local. So all we need to do is call `ctr.getPrice( )`, passing the `stock` and `currency` variables as parameters. Run `StockPriceApp2` and you'll get the following output:

```
Price is 75.33
```

We haven't yet looked at the SOAP messages transmitted by GLUE applications. Let's see how they differ from those transmitted by Apache SOAP. Here's the SOAP envelope that was transmitted by the GLUE APIs:

```
<soap:Envelope
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xmlns:xsd='http://www.w3.org/2001/XMLSchema'
 xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
 xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
 soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
  <soap:Body>
    <n:getPrice
       xmlns:n='http://tempuri.org/javasoap.book.ch4.StockPrice'>
         <arg0 xsi:type='xsd:string'>MINDSTRM</arg0>
         <arg1 xsi:type='xsd:string'>USD</arg1>
    </n:getPrice>
  </soap:Body>
</soap:Envelope>
```

The GLUE-generated SOAP envelope sets up the `xsi` and `xsd` namespace identifiers the same way as Apache SOAP. Instead of using the `SOAP-ENV` identifier, GLUE uses the name `soap`, which is hardly a significant difference; this is just a name. It can be called anything, as long as it is assigned the value of `http://schemas.xmlsoap.org/soap/envelope`. The `encodingStyle` attribute is the same as in the Apache SOAP examples as well. There seems to be an extra attribute, `soapenc`, that isn't referenced anywhere in the message, but that won't hurt anything. The `getPrice` element that represents the service method is not namespace-qualified by the service name, as is the case with Apache SOAP. The namespace is derived by

appending the class name of the service implementation class to the address *http://tempuri.org/*, which is the default namespace generated by the GLUE server. Unlike the Apache SOAP examples, the parameters are named `arg0`, `arg1`, etc. Since the method parameters are listed in the order they appear, the names are not significant. They are explicitly typed, just as they are in the Apache examples.

I prefer to use the actual service name to qualify the service method name, rather than an artificial service name derived by default. This is easy to correct; just supply a namespace property to the context object used when the server is executed. We did the same thing to specify the service activation mode. Here's a modified version of the server application (originally `CallCounterApp`), only this time we specify the namespace to use by calling `context.addProperty( )` with `"namespace"` as the first parameter and `"urn:StockPriceService"` as the second parameter. This new application publishes the `urn:StockPriceService`, so we don't need to go to the GLUE console.

```
package javasoap.book.ch4;
import electric.util.Context;
import electric.registry.Registry;
import electric.server.http.HTTP;
public class StockPriceApp3 {
   public static void main( String[] args )
      throws Exception {

      HTTP.startup("http://georgetown:8004/glue");
      Context context = new Context( );
      context.addProperty("activation", "application");
      context.addProperty("namespace", "urn:StockPriceService");
      Registry.publish("urn:StockPriceService",
         javasoap.book.ch4.StockPrice.class, context );
   }
}
```

If you still have an instance of `CallCounterApp` running, shut it down; it uses the same address and port number as this new application. Once you start `StockPriceApp3`, the namespace used to qualify the method name in SOAP messages will be `urn:StockPriceService`. Let's look at the response generated by the server for this request:

```
<soap:Envelope
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xmlns:xsd='http://www.w3.org/2001/XMLSchema'
 xmlns:soap='http://schemas.xmlsoap.org/soap/envelope/'
 xmlns:soapenc='http://schemas.xmlsoap.org/soap/encoding/'
 soap:encodingStyle='http://schemas.xmlsoap.org/soap/encoding/'>
   <soap:Body>
     <n:getPriceResponse xmlns:n='urn:StockPriceService'>
       <Result xsi:type='xsd:float'>75.33</Result>
     </n:getPriceResponse>
   </soap:Body>
 </soap:Envelope>
```

The response envelope is pretty much the same as the envelope generated by Apache SOAP; the only real difference is that the return value is contained in an element named `Result`, whereas in Apache SOAP this element is named `return`. But it's the structure of the response that's important, and as you can see the structure is the same as that of an Apache SOAP server's response.

4.7.1 Using in, out, and in/out Parameters

All of the examples so far have assumed that method parameters contain values to be passed from the client to the service, and that the service does not change the value of the parameters before returning. That's the default behavior of SOAP RPC services; the parameters in this case are referred to as `in` parameters because the value is passed "in" to the service. But SOAP also supports the use of two other parameter styles. Service methods may contain parameters that do not contain any data to be passed from the client to the service, but do contain data (to be read by the client) when the method returns. These are known as `out` parameters: the value of the parameter is passed "out" of the service. Parameters may also be `in/out`, where a value is passed in to the service and another value is passed out.

Not all SOAP implementation support the use of `out` or `in/out` parameters (Apache SOAP doesn't; GLUE does). That, by itself, should sound an alarm for you. It's not really a good idea to use features that are not widely supported unless you are in control of the servers as well as the clients of those servers. You may be able to avoid the need for using `out` or `in/out` parameters by defining a custom type that contains the values you want returned, and then defining your service methods to return an instance of that custom type. This is the approach I take; it also avoids any confusion about the purpose of the method. On the other hand, most recent implementations do support `in/out` parameters. In fact, many of them use the same technique that we're about to explore.

Now that I've cautioned against using them, let's take a look at how `out` and `in/out` parameters can be used. GLUE uses holder classes that encapsulate data types when using `out` and `in/out` parameters. There are already holder classes for the Java primitives and common classes like `java.lang.String`. For custom data types, GLUE includes a `holder` utility that generates holder classes for you. (Keep in mind that this approach is specific to GLUE; SOAP doesn't dictate the API, only the XML encoding.) Here's a modified version of the `StockPrice` class that includes two new methods. The first method is `getPriceOut( )`; it returns a `boolean` value indicating whether the request was successful. The first two parameters are the same as those in the original `getPrice( )` method: a stock symbol and the currency. The third parameter is called `price`; this is an `out` parameter that has no value when the method is invoked, and contains the resulting price when the method returns. So `getOutPrice( )` uses an `out` parameter to return the price, which it computes using the original `getPrice( )` method. The `price` parameter is an instance of `electric.util.holder.floatOut`, a holder class for floating-point `out` parameters.

The other new method is called `correctSymbol( )`, which converts a stock symbol to an appropriate format (all uppercase characters). It returns a `boolean` indicating success or failure, but more importantly uses a single `in/out String` parameter named `symbol`. This parameter contains a stock symbol when the method is called; when `correctSymbol( )` returns, it contains the corrected stock symbol. To support this behavior, the symbol parameter is declared as an instance of `electric.util.holder.java.lang.StringInOut`, a holder class for `String in/out` parameters. The `StockPriceApp3` can be used to publish the service without any modifications.

```
package javasoap.book.ch4;
import electric.util.holder.*;
import electric.util.holder.java.lang.*;
public class StockPrice {
   public float getPrice(String stock, String currency) {
      float result;
      // determine the price for stock and return it
      // in the specified currency
      result = (float)75.33;
      return result;
   }
   public boolean getOutPrice(String stock, String currency,
                     floatOut price) {
      try {
         price.value = getPrice(stock, currency);
         return true;
      }
      catch (Exception e) {
         return false;
      }
   }
   public boolean correctSymbol(StringInOut symbol) {
      symbol.value = symbol.value.toUpperCase(  );
      return true;
   }
}
```

Since we've modified the service by adding methods, we'll need to modify the `javasoap.book.ch4.IStockPriceService` interface in kind. Here's the new interface:

```
package javasoap.book.ch4;
import electric.util.holder.*;
import electric.util.holder.java.lang.*;
public interface IStockPriceService {
  float getPrice(String stock, String currency);
  public boolean getOutPrice(String stock, String currency,
                     floatOut price);
  public boolean correctSymbol(StringInOut symbol);
}
```

Now let's create an application to use this service. We'll call `correctSymbol( )` first to convert a stock symbol to uppercase using an `in/out` parameter, and then call `getOutPrice( )` to retrieve the price using an `out` parameter:

```
package javasoap.book.ch4;
import electric.registry.Registry;
import electric.registry.RegistryException;
import electric.util.holder.*;
import electric.util.holder.java.lang.*;
public class InOutApp {
   public static void main(String[] args) throws Exception
   {
      try {
         IStockPriceService ctr =
           (IStockPriceService)Registry.bind(
             "http://georgetown:8004/glue/urn:StockPriceService.wsdl",
             IStockPriceService.class);
         StringInOut stock = new StringInOut("MiNdStRm");
         boolean ok = ctr.correctSymbol(stock);
```

```
          if (ok) {
            System.out.println("Symbol converted to: "
                        + stock.value);
          }
          String currency = "USD";
          floatOut price = new floatOut(  );
          ok = ctr.getOutPrice(stock.value, currency, price);
          System.out.println("Result is " + price.value);
        }
        catch (RegistryException e)
        {
          System.out.println(e);
        }
    }
}
```

The stock symbol variable, `stock`, is instantiated with the symbol `"MiNdStRm"`. We then pass that variable, which is an instance of `electric.util.holder.java.lang .StringInOut`, to the `correctSymbol( )` method of the service interface. When the method call returns, we expect the symbol to have been converted to uppercase. Next we call `getOutPrice( )`, passing `stock.value` as the first parameter. The currency is passed as before, and an instance of `electric.holder.floatOut` called `price` is passed as the last parameter. When the method returns, `price` contains the stock price. Running the example produces this output:

```
Symbol converted to: MINDSTRM
Result is 75.33
```

As you can see, both the `out` and `in/out` parameters were handled as expected. Before we finish up, let's take a look at the parts of the SOAP envelopes passed between client and server that were affected by our use of `out` and `in/out` parameters. The call to `correctSymbol( )` results in a response envelope that looks a bit different from those we've seen before. The returned `boolean` value is immediately followed by the modified parameter value. Here's the SOAP body for that response:

```
<soap:Body>
  <n:correctSymbolResponse xmlns:n='urn:StockPriceService'>
    <Result xsi:type='xsd:boolean'>true</Result>
    <arg0 xsi:type='xsd:string'>MINDSTRM</arg0>
  </n:correctSymbolResponse>
</soap:Body>
```

The modified `in/out` parameter value in XML element `arg0` comes right after the `Result` element. And, of course, you can see that the original value has been converted to uppercase. Now let's look at part of the envelope sent to the server when `getOutPrice( )` was invoked. There are only two parameter values encoded, because the third parameter is an `out` parameter and so has no value at the time of the method invocation.

```
<n:getOutPrice xmlns:n='urn:StockPriceService'>
  <arg0 xsi:type='xsd:string'>MINDSTRM</arg0>
  <arg1 xsi:type='xsd:string'>USD</arg1>
</n:getOutPrice>
```

Finally, here's the response part of the SOAP body that was returned by the call to `getOutPrice( )`. The value of the `price` parameter follows the result itself. The first two

parameters don't have any values because they are both `in` parameters, and so don't have modified values.

```
<n:getOutPriceResponse xmlns:n='urn:StockPriceService'>
  <Result xsi:type='xsd:boolean'>true</Result>
  <arg2 xsi:type='xsd:float'>75.33</arg2>
</n:getOutPriceResponse>
```

In the next chapter, we'll begin investigating the more complex features and capabilities of SOAP and its implementations.